Moles: Tool-Assisted Environment Isolation with Closures

Jonathan de Halleux and Nikolai Tillmann

Microsoft Research
One Microsoft Way, Redmond WA 98052, USA
{jhalleux,nikolait}@microsoft.com

Abstract. Isolating test cases from environment dependencies is often desirable, as it increases test reliability and reduces test execution time. However, code that calls non-virtual methods or consumes sealed classes is often impossible to test in isolation. Moles is a new lightweight framework which addresses this problem. For any .NET method, Moles allows test-code to provide alternative implementations, given as .NET delegates, for which C# provides very concise syntax while capturing local variables in a closure object. Using code instrumentation, the Moles framework will redirect calls to provided delegates instead of the original methods. The Moles framework is designed to work together with the dynamic symbolic execution tool Pex to enable automated test generation. In a case study, testing code programmed against the Microsoft SharePoint Foundation API, we achieved full code coverage while running tests in isolation without an actual SharePoint server. The Moles framework integrates with .NET and Visual Studio.

1 Introduction

In software testing, it is often desirable to test individual components in isolation. It makes testing more robust and scalable. Especially in the context of unit testing, where the intention is to test a single unit of functionality at a time, all irrelevant environment dependencies should be mocked, or simulated, so that the unit tests run quickly and give deterministic results. (This is in contrast to integration testing, where the goal is to test an entire software system, including all environment dependencies, at the same time. Integration tests are usually neither quick, nor entirely reliable.)

However, many programming languages and runtime systems with static type checking and binding have language constructs that enforce the use of particular implementations. For example, in .NET, the code-under-test may call static or non-virtual methods, or it may consume sealed classes. In those cases, the standard .NET runtime does not provide any way to redirect calls for testing purposes. As a result, isolating the code for testing purposes becomes impossible.

In theory, the best solution to the problem is to refactor the code [3], introducing explicit interface boundaries and allowing different interface implementations. However, it is often not possible in practice to refactor existing legacy code, especially when it is provided by a third party.

We have developed a new lightweight framework called Moles to address this problem. The Moles framework allows the test-code to provide alternative implementations for non-abstract methods of any .NET type. Using code instrumentation, the Moles framework will redirect calls to an alternative implementation. An alternative implementation is given as a .NET delegate instance, which consists of a method together with a receiver object. This enables the direct use of closures in languages such as C# 2.0. The Moles framework generates *mole types* which provide explicit type-safe attachment points to register delegates as alternative implementations for every method of the code-under-test. As a welcome side effect of explicit type-safe attachment points, the developer is helped by the Visual Studio editor via automatic code completion when writing delegate attachments for any particular attachment point.

The Moles framework has been designed to work together with the dynamic symbolic execution [4] tool Pex [18,14] to enable automated test generation. Pex relies on tracing control- and dataflow at runtime, analyzing the test-code together with the code-under-test. Pex can only analyze .NET code. When an environment-facing component is invoked that is not implemented in .NET itself, then Pex cannot infer the constraints that reflect the behavior of that component, and as a consequence Pex cannot generate test cases that achieve high code coverage. Moles allow the tester to replace any environment-facing component at test time with .NET code that simulates its behavior, which in turn enables test case generation with Pex. The test cases generated by Pex do not depend on Pex anymore, but they can reproduce their results in isolation, using only Moles.

We have performed a case study, testing code programmed against the SharePoint Foundation [10] API using Moles and Pex, and we were able to achieve full code coverage while running the code in isolation without the need for an actual SharePoint server.

Moles supports .NET 2.0 and higher; Moles integrates with Visual Studio 2008 and higher. Moles is part of Pex, an incubation project for Visual Studio developed at Microsoft Research. Pex is available for academic use and for commercial evaluation.

The main contributions of this paper are:

- The description of a low-level, instrumentation-based detour approach.
- The description of a code generation framework that provides type-safe attachment points for detours in the form of delegates.
- A formalization of the semantics of attaching, detaching, and invoking detours, which enables the analysis of detours, in particular with symbolic execution frameworks.
- An evaluation of the feasibility of our approach in the context of a real-world application.

We will explain the basic idea behind the Moles framework along a simple example in Section 2. Moles leverages several features of .NET and C#, notably delegates, anonymous methods, lambda-expressions, and closures, which we will discuss in Section 3. We will describe in detail the implementation of Moles, and give a formal description of its semantics in Section 4. We describe a case study in which we applied Moles and Pex on a SharePoint application in Section 5. We discuss related work in Section 6, and conclude in Section 7.

2 Motivating Example

Consider a C# method that throws an exception on January 1st of 2000.

```
public static class Y2KChecker {
  public static void Check() {
    if (DateTime.Now == new DateTime(2000,1,1))
      throw new ApplicationException("y2k bug!");
  }
}
```

(Note: `Now` is a *property* of the `DateTime` type which is part of the .NET framework. When referencing this property as shown in the code above, a call to a method `get_Now` is generated by the C# compiler.)

Testing this method is particularly problematic because the program depends on `DateTime.Now`, a property that in turn depends on the computer clock. In other words, `DateTime.Now` is environment-dependent and non-deterministic. Moreover, the `DateTime.Now` property is static, so it is not possible to use virtual method overriding to supply a different implementation at test time. This problem is symptomatic of the isolation issue in unit testing: programs that directly call into the database APIs, communicate with web services, etc., are hard to unit test because their logic depends on the environment.

If .NET allowed us to redefine the meaning of static methods at runtime, we could easily solve this testing problem by replacing the computed value of `DateTime.Now` with something else.

```
// does not compile: DateTime.Now cannot be set
DateTime.Now = new DateTime(2000, 1, 1);
```

Moles enables this scenario. *Mole types* provide a mechanism to detour any .NET method. When a method is detoured, all invocations of that method get forwarded to an alternative implementation, and the original method is never actually invoked.

To this end, the Moles code generator creates a type `MDateTime` which has a settable property `NowGet`. This property can be set to an instance of a delegate which doesn't take parameters, and returns a `DateTime` value. C# 3.0 allows one to write a function inline within another method.

```
// attach delegate to the mole property to redirect DateTime.Now
// to return January 1st of 2000
MDateTime.NowGet = () => new DateTime(2000, 1, 1);
Y2KChecker.Check();
```

After setting `MDateTime.NowGet`, all calls to the property getter of `DateTime.Now` are detoured to the user-supplied delegate. This is realized by code instrumentation. To enable the code instrumentation when using the Visual Studio Unit Test framework, the attribute `[HostType("Moles")]` must be added to instruct the Visual Studio Unit Test framework to run the test under the control of the Moles instrumentation framework.

```
[TestMethod]
[HostType("Moles")] // run with code instrumentation
```

```
public void Y2kCheckerTest() {
  ...
}
```

The [TestMethod] annotation identified the above method as a unit test.

Pex, an automated white-box test generation tool for .NET, enables Parameterized Unit Testing [19]. The following method is identified as a parameterized unit test by the [PexMethod] attribute. It states that for all possible parameter values, the test should succeed (and not terminate with an uncaught exception).

```
[PexMethod]
public void Y2kCheckerTest(DateTime time) {
  // hook to the method to redirect DateTime.Now
  MDateTime.NowGet = () => time;
  Y2KChecker.Check();
}
```

By symbolically tracing the test inputs through the C# compiler-generated closure, the delegate, and the detoured invocation of DateTime.Now, Pex discovers the path condition time == new DateTime(2000,1,1) where time is the test input. As a result, Pex generates two traditional unit tests such as the two unit tests shown below, one where the path condition is fulfilled, and one where it is not.

```
[TestMethod]
[HostType("Moles")]
public void Y2kCheckerTest01() {
  Y2kCheckerTest(new DateTime(1, 1, 1));
}
[TestMethod]
[HostType("Moles")]
public void Y2kCheckerTest02() {
  Y2kCheckerTest(new DateTime(2000, 1, 1));
}
```

Without Moles, Pex would not have been able to construct test cases that cover all cases in the code, as the actual implementation of DateTime.Now relies on system calls to query the current machine time, which is outside of the semantics of .NET, and therefore outside of the scope which Pex can analyze. The resulting test cases can be executed in isolation, only relying on Moles to make the execution deterministic by circumventing the environment-facing components, without requiring symoblic execution with Pex.

3 Background: Delegates, Lambda-Expressions, and Closures

This section gives an overview of .NET delegates, C# anonymous methods, lambda-expressions, and closures, which the Moles framework leverages.

In .NET, *delegates* allow the representation and invocation of function pointers as values, in a fully type-safe manner. In addition to the actual function pointer, a *delegate value* may also contain an object reference, which serves as the first argument of the target function, usually the receiver object of an instance method. A *delegate type* specifies the return type and the parameter types of a method call. Delegate types are named. For example, the following declaration defines a delegate type named `D` with two integer parameters and a string return type. (As you can see, C# requires the specification of parameter names, but they are semantically irrelevant.)

```
delegate string D(int x, int y);
```

Consider a class with the following method.

```
string Compute(int x, int y) {
  return (x+y).ToString();
}
```

Creating an instance of a delegate is written similar to a constructor call in C#; the parameter is the name of a method, qualified with an expression holding the object reference that should serve as the receiver object for an instance method (and which is omitted when referring to a static method):

```
D d = new D(this.Compute);
```

In C#, the syntax to invoke a delegate is similar to the syntax for invoking a regular method. For the example above, the delegate invocation `d(23,42)` will in effect invoke `this.Compute(23,42)`.

C# supports so-called *anonymous methods* starting with version 2.0. This allows the inlining of the above `Compute` method:

```
D d = delegate(int x, int y) {
  return (x+y).ToString();
};
```

Starting from version 3.0, C# supports so-called *lambda-expressions* which allow further abbreviating the code to the following equivalent code:

```
D d = (x, y) => (x+y).ToString();
```

An anonymous method or a lambda-expression may refer to local variables defined in the scope of the outer method.

```
int offset = 42;
D d = (x, y) => (x + y + offset).ToString();
```

In this case the C# compiler will generate a closure class which will hold all the referenced local variables as fields, and the C# compiler will take care of redirecting all accesses to an instance of the closure class. The following code represents the equivalent code generated by the C# compiler:

```
Closure c = new Closure();
c.offset = 42;
D d = new D(c.Body);

class Closure {
  public int offset;
  public string Body(int x, int y) {
    return (x + y + offset).ToString();
  }
}
```

In C#, the body of the anonymous method or lambda-expression may mutate the captured local variables, and the changes will be visible to all other references to the same variable. In other words, closures in C# capture variables, not just values.

4 Our Approach: Moles

4.1 Code Instrumentation, and Low-Level Detours Library

The Moles framework builds on top of the Extended Reflection code instrumentation framework to realize the detouring of method calls. Extended Reflection is also used by Pex and CHESS[11]. The details of the code instrumentation are hidden from the user who interacts only with the generated high-level *mole types* described in Section 4.2.

The instrumentation is done as follows. At the beginning of each method some code is inserted. This code queries a low-level detours library as to whether any detour was attached to this particular method. If so, then the detour delegate is invoked, otherwise, the normal code is executed.

For example, consider the following method.

```
string Compute(int x, int y) {
  return (x + y).ToString();
}
```

After instrumentation, the code will be similar to the following pseudo-code.

```
string Compute(int x, int y) {
  // obtain identifier of this method
  Method thisMethod = this.GetType().GetMethod(
    "Compute", new Type[]{typeof(int), typeof(int)});
  // query if a detour has been attached
  Delegate detour = _Detours.GetDetour(this, thisMethod);
  if (detour != null) {
    // pseudo-code; actual implementation avoids boxing
    return (string)_Detours.InvokeDetour(
      detour, this,
      new object[] { x, y });
  }
  // else execute normal code
  return (x + y).ToString();
}
```

The above code uses a low-level detours library shown in Figure 1, which manages all attached detours, and provides facilities to query and invoke detour delegates.

While the above code accurately reflects the behavior after code instrumentation has been applied, note that our actual code instrumentation results in more efficient code that avoids casting and boxing by using (potentially) “unsafe” .NET instructions (which are guaranteed to be “safe” in the particular way they are used by Moles), in effect simply passing through the values. This is possible as the instrumented code may bypass type-safety checks at test time.

```
public class _Detours {
  // attaching and detaching
  public static void AttachDetour(
    object receiver, Method method,
    Delegate detourDelegate) {
    ...
  }
  public static void DetachDetour(
    object receiver, Method method) {
    ...
  }
  // quering and invoking
  public static Delegate GetDetour(
    object receiver, Method method) {
    ...
  }
  public static object InvokeDetour(
    Delegate detour,
    object receiver, object[] args) {
    ...
  }
}
```

Fig. 1. Detours library used by Moles

4.2 Mole Type Generation for Type-Safety

When attaching and detaching moles via our general detours library shown in Figure 1, type-safety cannot be guaranteed at compile time: The `AttachDetour` method takes a method identifier, and a delegate of type `Delegate`, which is the base type of all delegate types. In order to enforce that attaching (and detaching) of moles is always done in a type-safe manner, the Moles framework generates specialized code. Besides type-safety, the specialized code also frees the developer from constructing method identifiers using the .NET reflection API, and it enables Visual Studio editor support for automatic code completion when attaching or detaching moles.

In the following, we discuss the details of the Moles code generator, which systematically covers static methods and constructors, as well as instance methods, in an instance-agnostic and an instance-specific way.

For every type t, a mole type `M`t is generated. For every method in t, including the implicit getter and setter methods for .NET properties, and adders and removers for .NET events, settable mole properties are generated as explained in the following. The generated type Mt is placed into a sub-namespace of type t; the suffix .Moles is added to the namespace of t. The generated types are placed in a separate *assembly*, which is a collection of types in .NET that can be addressed independently.

Static Methods and Constructors. For each static method in type t with parameter types $T_1, T_2, \ldots, T_n$ and return type U, a settable static property in type Mt is generated, with a delegate type Func similar[1] to the following:

$$\texttt{delegate}\ U\ \texttt{Func}\ (T_1, T_2, \ldots, T_n)$$

The same applies to instance constructors, where the implicit this argument becomes the first parameter T_1. The name of the property starts with the name of the static method, appended by short type names of the parameters, and possibly a number to make all generated property names distinct. This allows distinguishing different overloads with the same method name.

Example: The following pseudo-code illustrates the generated implementation of MDateTime.NowGet that we used in the motivating example in Section 2. (The first argument to AttachDetour and DetachDetour is null, as DateTime.Now is a static property that does not require any instance.) The code uses the low-level detours library shown in Figure 1.

```
static Func/*delegate DateTime Func()*/ NowGet {
  set { // property setter has implicit 'value' parameter
    Method method = typeof(DateTime).GetMethod("get_Now");
    if (value == null)
      _Detours.DetachDetour(null, method);
    else
      _Detours.AttachDetour(null, method, value);
  }
}
```

Instance Methods (for all instances). For each instance method in type t with explicit parameter types $T_1, T_2, \ldots, T_n$ and return type U, a settable static property in the nested type Mt.AllInstances is generated, with a delegate type Func similar to the following:

$$\texttt{delegate}\ U\ \texttt{Func}\ (t, T_1, T_2, \ldots, T_n)$$

Note the first parameter type, which represents the previously implicit this argument of the instance method.

Example: The following pseudo-code shows the implementation of the generated static property ComputeInt32Int32 for the Compute example earlier in this section. Note that the name consists of the original method name, Compute, appended by Int32Int32, which represents the two int parameters (whose canonical .NET type name is System.Int32):

[1] Instead of generating a separate delegate type for each moled method, generic delegate types are instantiated. However, to simplify the exposition, we do not discuss .NET generics in this paper.

```
static Func/*delegate string Func(int,int)*/ ComputeInt32Int32 {
  set { // property setter has implicit 'value' parameter
    Method method = typeof(ComputeType).GetMethod("Compute",
      new Type[] { typeof(int), typeof(int) });
    if (value == null)
      _Detours.DetachDetour(null, method);
    else
      _Detours.AttachDetour(null, method, value);
  }
}
```

The first argument to `AttachDetour` and `DetachAttach` is `null`, indicating that this detour should apply to all instances.

Instance Methods (for a specific instance). As an alternative to detouring all calls to instance methods regardless of the implicit `this` parameter, detouring can dispatch to different detour delegates depending on the value of the `this` parameter. To this end, the low-level API maintains a mapping of receiver-object and method pairs to detour delegates.

The high-level mole types allow binding to specific receiver objects by making Mt instantiable, associating each instance of Mt with an instance of t, and by a providing settable instance properties on Mt for all instance methods of t.

For each instance method in type t with parameter types $T_1, T_2, \ldots, T_n$ and return type U, a settable instance property in type Mt is generated, with a delegate type similar to the following:

$$\texttt{delegate}\, U\ \texttt{Func}\ (T_1, T_2, \ldots, T_n)$$

Note that unlike the static property in the nested `AllInstances` type, there is no provision to pass on the implicit `this` argument of the instance method. (However, it could be captured in the closure object of the detour delegate.)

The Mt type has an instance property called `Instance` to access the associated t instance. The Mt type also defines an implicit conversion operator to t.

Example: The following pseudo-code shows the implementation of the generated property `ComputeInt32Int32` for the `Compute` example earlier in this section.

```
Func/*delegate string Func(int,int)*/ ComputeInt32Int32 {
  set { // property setter has implicit 'value' parameter
    Method method = typeof(ComputeType).GetMethod("Compute",
      new Type[] { typeof(int), typeof(int) });
    if (value == null)
      _Detours.DetachDetour(this.Instance, method);
    else
      _Detours.AttachDetour(this.Instance, method, value);
  }
}
```

The first argument to `AttachDetour` and `DetachDetour` is `this.Instance`, indicating that this detour should apply to only to the specific instance associated with the current mole instance.

4.3 Formalization

We discussed the code instrumentation and the generated type-safe mole types, and illustrated our implementation using the low-level `_Detours` library. The Moles framework was motivated by Pex [18,14], an automated white-box test generation tool, which uses dynamic symbolic execution [4]. Pex tracks precisely the control- and data-flow of program executions, in order to gather *path conditions* describing the condition, expressed as a formula over the test inputs, under which the program takes a particular execution path. Variations of such path conditions are then solved by an automated constraint solver, in order to compute new test inputs that will exercise a new execution path. Symbolic execution requires a precise model of the semantics of the operations of the executing program in order to build exact path conditions. In effect, Moles adds new and modifies existing operations of the execution environment, which must be modeled precisely: Moles adds the ability to attach and detach detour delegates, and Moles modifies the meaning of a method call: If a detour delegate is currently attached, control is transferred to the detour delegate, otherwise the method executes normally.

Detour delegates are effectively a pair, consisting of a closure object, and a pointer to a function. If the detoured method takes a receiver object argument, then a detour can either apply to all receiver objects, or only to a particular one.

In the following, we describe the meaning of some basic operations, attaching and detaching detour delegates, method calls and call returns.[2]

We consider the following operations. All instructions belong to the sort $INSTR$, the names x, y, $y_1, y_2, \ldots, y_n$ refer to local variables of sort VAR, and M and N are method identifiers of sort $PROC$.
General computation, assignment and conditional control flow:[3]

- Assigning a value computed by a built-in function f:
  `assign` $x := f(y_1, y_2, \ldots, y_n)$
- Conditional branching:
  `if`(x) `goto` PC

For attaching and detaching moles:

- Attaching a mole closure (y, N) for a method M:
  `mole` $M := (y, N)$
- Attaching a mole closure (y, N) for a specific instance x and its method M:
  `mole` $(x, M) := (y, N)$
- Detaching a mole for a method M:
  `unmole` M
- Detaching a mole for a specific instance x and its method M:
  `unmole` (x, M)

[2] We ignore many details of the execution engine which are not relevant here, including object allocation, object field accesses, virtual method resolution, etc.

[3] We include these instruction for illustrative purposes; they are not necessary to understand the behavior of Moles.

For calling and returning:

– Calling a method M with arguments $y_1, y_2, \ldots, y_n$ and storing the result in x:
  $x := \texttt{call}\, M(y_1, y_2, \ldots, y_n)$
  (Note in this operation, any initial `this` argument is made explicit.)
– Returning from a method call with value y:
  $\texttt{ret}\; y$

A program P has a function $\mathit{start}_P : \mathit{PROC} \rightarrow \mathcal{N}$ which maps a method to the offset of its first instruction, and a function $\mathit{instr}_P : \mathcal{N} \rightarrow \mathit{INSTR}$ which maps each offset to its associated instruction. A program starts at offset 0.

The program semantics are described by the constant $\mathit{init}_P : \mathit{HEAP} \times \mathit{STACK}$, which represents the initial program state, together with the function $\mathit{step}_P : \mathit{HEAP} \times \mathit{STACK} \rightarrow \mathit{HEAP} \times \mathit{STACK}$ which represents the execution of a single instruction by transforming the program heap and stack. The heap maps values to values, and the stack is a sequence of stack frames, where each stack frame is given by a pair of a current program counter, and a mapping of local variables of sort VAR to values. While in general the heap may hold other objects or global state, we only use it to model the globally active attached detours. Every method has a special set of local variables for incoming arguments: arg_1, arg_2, ...

We use standard notations for maps and sequences. We write $\{\mapsto\}$ for the empty map, $\{x \mapsto y\}$ for the map with one entry that maps x to y, $M \oplus N$ for the combined map where N takes precedence over M, and $M \setminus x$ for the map that is M except that it does not contain index x, and $\{x \mapsto y : p\}$ for map that maps x to y wherever p holds, and $x \mapsto y \in M$ holds if the map contains the given mapping, $M(x)$ represents the value of M at index x. We write $[x]$ for the sequence holding one element x, and $x +\!\!+\, y$ for the concatenation of sequences x and y.

Initial program state:
$$\mathit{init}_P = (\{\mapsto\}, [(0, \{\mapsto\})])$$

General computation, assignment and conditional control-flow:

$$\begin{aligned}
&\mathit{step}_P(H, R +\!\!+ (PC, L))\\
&\qquad \text{where } (\mathit{instr}_P(PC) = \texttt{assign}\, x := f(y_1, y_2, \ldots, y_n))\\
&\quad = (H, R +\!\!+ (PC + 1, L \oplus \{x \mapsto \hat{f}(L(y_1, y_2, \ldots, y_n))\}))\\
&\qquad \text{where } \hat{f} \text{ is the built-in function computing } f\\
&\mathit{step}_P(H, R +\!\!+ (PC, L))\\
&\qquad \text{where } (\mathit{instr}_P(PC) = \texttt{if}(x)\ \texttt{goto}\ PC')\\
&\quad = \begin{cases} (H, R +\!\!+ (PC', L)) & \text{if } L(x) \text{ is true}\\ (H, R +\!\!+ (PC + 1, L)) & \text{otherwise}\end{cases}
\end{aligned}$$

In words: If the current instruction is an assignment, then it does not change the heap, it increments the current program counter, and it assigns the value, computed over the local variables values, to a local variable. If the current instruction is a conditional control-flow operation, it evaluates the given condition with the current local variables assignment, and if it evaluates to `true`, the program counter changes to the specified target, otherwise it is incremented by one.

For attaching and detaching moles:

$$\begin{aligned}
&\mathit{step}_P(H, R +\!\!+ (PC, L)) \\
&\qquad \text{where } (\mathit{instr}_P(PC) = \texttt{mole}\ M := (y, N)) \\
&\quad = (H \oplus \{M \mapsto (y, N)\}, R +\!\!+ (PC + 1, L)) \\
&\mathit{step}_P(H, R +\!\!+ (PC, L)) \\
&\qquad \text{where } (\mathit{instr}_P(PC) = \texttt{unmole}\ M) \\
&\quad = (H \setminus M, R +\!\!+ (PC + 1, L)) \\
&\mathit{step}_P(H, R +\!\!+ (PC, L)) \\
&\qquad \text{where } (\mathit{instr}_P(PC) = \texttt{mole}\ (x, M) := (y, N)) \\
&\quad = (H \oplus \{(x, M) \mapsto (y, N)\}, R +\!\!+ (PC + 1, L)) \\
&\mathit{step}_P(H, R +\!\!+ (PC, L)) \\
&\qquad \text{where } (\mathit{instr}_P(PC) = \texttt{unmole}\ (x, M)) \\
&\quad = (H \setminus (x, M), R +\!\!+ (PC + 1, L))
\end{aligned}$$

The above instructions basically add moles to or remove moles from the global state, represented here as H.

For calling and returning:

$$\begin{aligned}
&\mathit{step}_P(H, R +\!\!+ (PC, L)) \\
&\qquad \text{where } (\mathit{instr}_P(PC) = \_ := \texttt{call}\ M(y_1, \ldots, y_n)) \\
&\quad = \begin{cases} (H, R +\!\!+ (PC, L) +\!\!+ f^{\text{One}}_{v,N}) \text{ if } (L(y_1), M) \mapsto (v, N) \in H \\ (H, R +\!\!+ (PC, L) +\!\!+ f^{\text{All}}_{v,N}) \text{ if } M \mapsto (v, N) \in H \\ (H, R +\!\!+ (PC, L) +\!\!+ f) \quad \text{otherwise} \end{cases} \\
&\qquad \text{where the new frame } f_{v,N}, f_N \text{ or } f \text{ is defined as follows:} \\
&\qquad f^{\text{One}}_{v,N} = (\mathit{start}_P(N), \{\mathit{arg}_1 \mapsto v\} \oplus \{\mathit{arg}_i \mapsto L(y_i) : 2 \le i \le n\}) \\
&\qquad f^{\text{All}}_{v,N} = (\mathit{start}_P(N), \{\mathit{arg}_1 \mapsto v\} \oplus \{\mathit{arg}_{i+1} \mapsto L(y_i) : 1 \le i \le n\}) \\
&\qquad f \quad = (\mathit{start}_P(M), \{\mathit{arg}_i \mapsto L(y_i) : 1 \le i \le n\}) \\
&\mathit{step}_P(H, R +\!\!+ (PC, L) +\!\!+ (PC', L')) \\
&\qquad \text{where } (\mathit{instr}_P(PC) = x := \texttt{call}\ \_(\_) \quad \wedge \quad \mathit{instr}_P(PC') = \texttt{ret}\ y) \\
&\quad = (H, R +\!\!+ (PC + 1, L \oplus \{x \mapsto L'(y)\}))
\end{aligned}$$

In words: If the current instruction is a call operation, it is checked if an instance-specific mole, an instance-agnostic mole, or no mole has been attached. In any case, a new stack frame is added. In the case of an instance-specific mole, the stack frame $f^{\text{One}}_{v,N}$ points the mole method, the first argument is the mole closure, the instance is dropped, and all other arguments are passed on. In the case of an instance-agnostic mole, the stack frame $f^{\text{All}}_{v,N}$ points to the mole method, the first argument is the mole closure, and all other arguments are passed on, but are shifted by one. Otherwise, a regular new stack frame is created, whose initial local variables arg_i map to the specified local values in the current stack frame. If the current instruction is a return operation (and the instruction in the previous stack frame is a call operation), the current stack frame is removed, and the current result value is stored in the previous stack frame.

5 Case Study

5.1 Introduction to SharePoint Unit Testing

We applied Moles and Pex to test a Microsoft SharePoint application. Microsoft SharePoint Foundation [10] provides a platform for comprehensive content management,

```
public void UpdateTitle(SPItemEventProperties properties) {
  using (SPWeb web = new SPSite(properties.WebUrl).OpenWeb()) {
    SPList list = web.Lists[properties.ListId];
    SPListItem item = list.GetItemById(properties.ListItemId);
    string contentType = (string)item["ContentType"];
    if (contentType.Length < 5)
      throw new ArgumentException("too short");
    if (contentType.Length > 60)
      throw new ArgumentOutOfRangeException("too long");
    if (contentType.Contains("\r\n"))
      throw new ArgumentException("no new lines");
    item["Title"] = contentType;
    item.SystemUpdate(false);
  }
      }
```

Fig. 2. SharePoint code-under-test

enterprise search, collaboration, communication, and more. SharePoint Foundation provides an API that allows building applications on top of the platform. It is often difficult to create unit tests for such applications as it is impossible to execute the underlying SharePoint Object Model without being connected to a live SharePoint site that runs the SharePoint server software. For that reason, most of the “unit tests” that have been written in the past for SharePoint applications are actually integration tests because they need a live system to run. As a result, those test take a long time to run which defeats the goals of unit testing: the tests should run fast, test only one component, and be completely reproducible.

Furthermore, most test cases induce some state change in the SharePoint server, which either means that state-dependent test cases are unreliable, or that the SharePoint server and its database must be restarted and reinitialized after every individual test case, which again contributes to long running test cases.

The SharePoint Object Model, including classes such as `SPSite`, `SPWeb`, . . ., does not allow the tester to inject fake service implementations because most of the SharePoint classes are sealed types with non-public constructors. We will show how to use the Moles framework for detouring and to inject a fake service implementation.

5.2 The Code-under-Test

In this case study, we will test a simple SharePoint application that updates a field in a `SPListItem` instance, after performing some validation of the supplied values. The method shown in Figure 2 performs the actual work.

The above `UpdateTitle` method presents the typical challenges of unit testing with SharePoint Foundation: it connects to a live `SPWeb` through a `SPSite`. Unfortunately, it is not possible to provide a fake implementation of `SPSite` since this class is sealed.

The method under test is encapsulated into a `SPItemEvenReceiver` implementation, shown below.

```
public class ContentTypeItemEventReceiver
  : SPItemEventReceiver {
  public override void ItemAdded(
    SPItemEventProperties properties) {
    this.EventFiringEnabled = false;
    this.UpdateTitle(properties);
    this.EventFiringEnabled = true;
  }
  ...
}
```

5.3 The (Parameterized) Unit Test with Moles

The choice of `contentType` is critical to test the implementation of the `UpdateTitle` method, as different values take different code paths in the application. Pex is a tool that automates the process of finding relevant test inputs in a systematic way. Pex executes the code-under-test while monitoring every instruction executed by the .NET code. In particular, Pex records all the conditions that the program checks along the executed code path, and how they relate to the test input. Pex gives this information to a constraint solver, which then crafts new inputs that will trigger different code paths. Explaining how Pex works in more detail is beyond the scope of this paper, so we refer the reader to the Pex documentation for further details [14,18].

We write a parameterized unit test, shown in Figure 3, that states that the method under test should work correctly for all values of `listItemId` and `contentType`. The test constructs moles in a fashion mirroring the usage of the SharePoint classes in the code-under-test: First, the `WebUrl` property is queried from an `SPItemEvent-Properties` instance (and later the `ListId` and `ListItemId` properties). Then the constructor of `SPSite` is called, and then its method `OpenWeb`, and so on. We use the C# 3.0 *object initializer* syntax, where `T t = new T(); t.P = x;` can be abbreviated to `new T() { P = x; }`. Also note that the names of the mole properties for the indexer accesses `_[_]` in the code-under-test are `ItemGetString` and `ItemSetString`.

The `contentType` parameter is passed through (via closures), until it is returned by `ItemGetString`, and we also implemented logic to assert that the correct string is set in the call to `ItemSetStringObject`, and we maintain state in the `titleSet` variable to check later on that this call actually happened. The assignment `titleSet = true` is done in a nested anonymous method, and it is only due to the fact that closures in C# capture variables, and not just values, that we can query this modification in the assertion at the end: `Assert.IsTrue(titleSet)`.

While the test case seems to be quite long, this has to be seen in relation to effort it takes to setup and maintain a SharePoint server with appropriate test data.

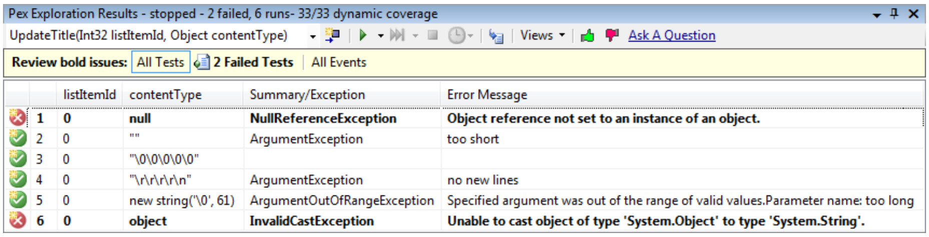
5.4 Evaluation

We let Pex explore the parameterized unit test code together with the code-under-test. Within seven seconds[4], the following six (non-parameterized) unit tests are generated.

[4] Pex and Moles version 0.21, Intel Core Duo P9500 with 2.53Ghz, 4GB RAM.

```
[PexMethod]
public void UpdateTitle(int listItemId, object contentType) {
  //// arrange ////
  string url = "http://foo";
  Guid listId = new Guid();
  var properties = new MSPItemEventProperties {
    WebUrlGet = () => url,
    ListIdGet = () => listId,
    ListItemIdGet = () => listItemId
  };
  bool titleSet = false;
  MSPSite.ConstructorString = (site, _url) => {
    new MSPSite(site) {
      OpenWeb = () => new MSPWeb {
        Dispose = () => { },
        ListsGet = () => new MSPListCollection {
          ItemGetGuid = id => {
            Assert.IsTrue(listId == id);
            return new MSPList {
              GetItemByIdInt32 = itemId => {
                Assert.IsTrue(listItemId == itemId);
                return new MSPListItem {
                  InstanceBehavior =
                    MoleBehaviors.DefaultValue,
                  ItemGetString = name => {
                    Assert.IsTrue("ContentType" == name);
                    return contentType;
                  }, // ItemGetString
                  ItemSetStringObject = (name, value) => {
                    Assert.IsTrue("Title" == name);
                    Assert.IsTrue(contentType == value);
                    titleSet = true;
                  } // ItemSetStringObject
                }; // new MSPListItem
              } // GetItemByIdIn32
            }; // new MSPList()
          } // ItemGetGuid
        } // new MSPListCollection
      } // new MSPWeb
    }; // new MSPSite
  }; // MSPSite.New
  //// act ////
  var target = new ContentTypeItemEventReceiver();
  target.UpdateTitle(properties);
  //// assert ////
  Assert.IsTrue(titleSet);
}
```

Fig. 3. Parameterized Unit Test for `UpdateTitle`



Pex Exploration Results - stopped - 2 failed, 6 runs- 33/33 dynamic coverage

UpdateTitle(Int32 listItemId, Object contentType)

		listItemId	contentType	Summary/Exception	Error Message
✗	**1**	**0**	**null**	**NullReferenceException**	**Object reference not set to an instance of an object.**
✓	2	0	""	ArgumentException	too short
✓	3	0	"\0\0\0\0\0"		
✓	4	0	"\r\r\r\r\n"	ArgumentException	no new lines
✓	5	0	new string('\0', 61)	ArgumentOutOfRangeException	Specified argument was out of the range of valid values.Parameter name: too long
✗	**6**	**0**	**object**	**InvalidCastException**	**Unable to cast object of type 'System.Object' to type 'System.String'.**

Fig. 4. Screenshot of the Pex tool with generated test cases

Figure 4 shows a screenshot of the Pex tool with the results. In the table, there is one row per generated test case; the input data for each generated test case is shown in the columns `listItemId` and `contentType`. (In addition to the table, Pex also generated C# code for the test cases.) When an exception occurred, then the `Summary/Exception` and `Error Message` columns are filled. We configured Pex to treat an `Argument...Exception` as permissible behaviors. All tests cases but two are passing: One failing test case causes a `NullReferenceException`, and the other an `InvalidCastException`. Those are unexpected exceptions. In addition to revealing the two unexpected exceptions, the generated test suite covers all explicit branches in the code, and as a consequence all basic blocks, in the test case as well as the code-under-test, We confirmed this result via the coverage reports that Pex generates as a side effect, as well as independently using the Visual Studio code coverage analysis.

While test generation in isolation with Pex and Moles took seven seconds, starting a live SharePoint server, and bringing it and its database back to a well-defined initial state for unit testing purposes takes about an order of magnitude longer.

5.5 Reproduction of Case Study

A detailed tutorial to reproduce the results of this case study can be found on the Pex website [15].

6 Related Work

The basic idea of dynamic detouring for testing purposes is not new. For .NET, a commercial tool is TypeMock Isolator [20]. Several detouring frameworks are available for Java, notably JMockit [7]. However, both TypeMock Isolator and JMockit focus not on providing basic or systematic detouring functionality, but instead they are *mocking* frameworks. This means that they provide powerful APIs that allow stating expectations, imposing verification checks, and adding annotations to existing methods. Moles in contrast simply provides a light-weight delegate-based detouring facility, which other mock frameworks do not provide. Simplicity is paramount for automated test generation via dynamic symbolic execution, as the semantics of all operations must be modeled precisely.

General detour mechanisms, which are not specialized for unit testing, exist for other platforms as well, e.g. for Win32 [6]. While our Moles framework has been implemented for the .NET framework, the idea of generating type-safe attachment points can be applied to any other platform where a general (untyped) detour mechanism is available.

Moles enables effective stubbing of legacy code [2], requiring a developer or tester to manually write code. There are several approaches to automatically turning slow system tests into focused unit tests [17,21,1,8,12]. They all require existing system tests, and are strictly capture and replay approaches. Some are implemented as a program transformation at the bytecode level, other at the source code level. But in any case, they do not provide an easy mechanism to attach custom behaviors in user-written test code to the detoured environment interactions.

Some approaches to represent mock objects are based on instrumentation via aspect-oriented programming (AOP) [16], which can also be used to automatically turn slow system tests into focused unit tests [13]. While AOP can certainly be used to realize the instrumentation necessary to detour method calls, the syntax and semantics typically employed by general-purpose AOP languages add a considerable complexity to the semantics of a unit test.

Isolation for testing is related to the idea of encapsulation, which in principle should allow the treatment of components as black boxes which can be swapped out at test time. Encapsulation is a key principle of the object-oriented software paradigm [9], and was already proposed earlier in the form of abstract data types [5]. While a consequent application of these ideas should make the problem of mocking for unit testing go away, legacy code does exist that violates those principles.

7 Conclusion and Future Work

We have developed a new lightweight framework called Moles which allows one to isolate unit tests by detouring calls to environment-dependent methods via .NET delegates. The changes to the operational semantics of a program are easy to model formally, which enables automated program analysis. We have performed a case study where we applied Moles and Pex, an automated test generation tool based dynamic symbolic execution, on code that uses the SharePoint Foundation API. After isolating all dependencies, multiple test cases were automatically generated in seconds, while starting and resetting a real SharePoint server takes an order of magnitude longer. Moles is part of Pex, an incubation project for Visual Studio developed at Microsoft Research. Pex is available for academic use and for commercial evaluation.

Unlike traditional mock-based unit testing, unit tests with Moles do not impose a simple capture-replay structure where expectations and verification conditions on method calls are stated sequentially, without the ability to programmatically simulate the detoured behavior, or to maintain or manipulate state across method calls. Instead, by using delegates and C# anonymous methods, Moles makes it easy to write light-weight and reusable *models* of the original detoured methods. Closures in C# capture variables, not just values, which enables writing stateful models. Future work includes adding facilities to Moles which further simplify writing models by a skilled tester or developer.

References

1. Elbaum, S., Chin, H.N., Dwyer, M.B., Dokulil, J.: Carving differential unit test cases from system test cases. In: Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering, pp. 253–264. ACM, New York (2006)
2. Feathers, M.: Working Effectively with Legacy Code. Prentice Hall PTR, Englewood Cliffs (September 2004)
3. Fowler, M.: Refactoring: Improving the Design of Existing Code. Addison-Wesley, Reading (1999)
4. Godefroid, P., Klarlund, N., Sen, K.: DART: directed automated random testing. SIGPLAN Notices 40(6), 213–223 (2005)
5. Guttag, J.V., Horning, J.J.: The algebraic specification of abstract data types. Acta Informatica 10, 27–52 (1978)
6. Hunt, G., Brubacher, D.: Detours: binary interception of win32 functions. In: WINSYM'99: Proceedings of the 3rd conference on USENIX Windows NT Symposium, Berkeley, CA, USA, pp. 14. USENIX Association (1999)
7. JMockit developers. The JMockit testing toolkit (January 2010), `http://jmockit.googlecode.com/svn/trunk/www/about.html`
8. Joshi, S., Orso, A.: SCARPE: A technique and tool for selective record and replay of program executions. In: Proceedings of the 23rd IEEE International Conference on Software Maintenance (ICSM 2007), Paris, France (October 2007)
9. Micallef, J.: Encapsulation, reusability and extensibility in object-oriented programming languages. Journal of Object-Oriented Programming, 12–36 (April/May 1988)
10. Microsoft. Windows sharepoint services 3.0. (January 2010), `http://technet.microsoft.com/en-us/windowsserver/sharepoint/default.aspx`
11. Musuvathi, M., Qadeer, S., Ball, T., Basler, G., Nainar, P.A., Neamtiu, I.: Finding and reproducing heisenbugs in concurrent programs. In: OSDI, pp. 267–280 (2008)
12. Orso, A., Joshi, S., Burger, M., Zeller, A.: Isolating relevant Component Interactions with JINSI. In: Proceedings of the Fourth International ICSE Workshop on Dynamic Analysis (WODA 2006), Shanghai, China, May 2006, pp. 3–9 (2006)
13. Pasternak, B., Tyszberowicz, S., Yehudai, A.: GenUTest: a unit test and mock aspect generation tool. Int. J. Softw. Tools Technol. Transf. 11(4), 273–290 (2009)
14. Pex development team. Pex (2008), `http://research.microsoft.com/Pex`
15. Pex development team. Unit testing SharePoint Foundation with Microsoft Pex and Moles (April 2010), `http://research.microsoft.com/pex/pexsharepoint.pdf`
16. Rho, T., Kniesel, G.: Uniform genericity for aspect languages. Technical report, Needs, Options and Challenges, Special issue of L'Objet. Hermes Science Publishing (2004)
17. Saff, D., Artzi, S., Perkins, J.H., Ernst, M.D.: Automatic test factoring for Java. In: ASE '05: Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering, pp. 114–123. ACM Press, New York (2005)
18. Tillmann, N., de Halleux, J.: Pex - white box test generation for .NET. In: Beckert, B., Hähnle, R. (eds.) TAP 2008. LNCS, vol. 4966, pp. 134–153. Springer, Heidelberg (2008)
19. Tillmann, N., Schulte, W.: Parameterized unit tests. In: Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 253–262. ACM, New York (2005)
20. TypeMock Development Team. Isolator, `http://www.typemock.com/learn_about_typemock_isolator.html`
21. Xu, G., Rountev, A., Tang, Y., Qin, F.: Efficient checkpointing of java software using context-sensitive capture and replay. In: ESEC-FSE '07: Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering, pp. 85–94. ACM, New York (2007)